(72) Inventors:
• Fouladi, Farhad
Palo Alto, CA 94301 (US)
• Van Hook, Timothy J.
Atherton, CA 94027 (US)
• Cheng, Howard H.
Redmond, Washington 98053 (US)
• Moore, Robert
Redmond, Washington 98053 (US)

(74) Representative:
Horton, Andrew Robert Grant et al
BOWLES HORTON
Felden House
Dower Mews
High Street
Berkhamsted Hertfordshire HP4 2BL (GB)

(54) **Vertex cache for 3D computer graphics**

(57)   In a 3D interactive computer graphics system such as a video game display system, polygon vertex data is fed to the display engine via a vertex cache used to cache and organize indexed primitive vertex data streams. The vertex cache may be a small, low-latency cache memory local to the display engine hardware. Polygons can be represented as indexed arrays, e.g., indexed linear lists of data components representing some feature of a vertex (for example, positions, colors, surface normals, or texture coordinates). The vertex cache can fetch the relevant blocks of indexed vertex attribute data on an as-needed basis to make it available to the display processor -- providing spatial locality for display processing without requiring the vertex data to be prestored in display order. Efficiency can be increased by customizing and optimizing the vertex cache and associated tags for the purpose of delivering vertices to the graphics engine -- allowing more efficient prefetching and assembling of vertices than might be possible using a general-purpose cache and tag structure.
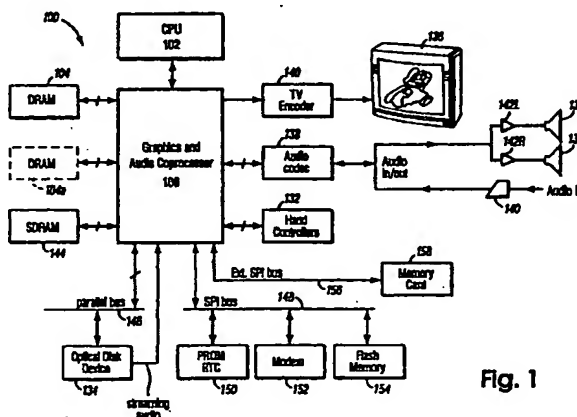
Fig. 1

EP 1 096 427 A2

Description

FIELD OF THE INVENTION

[0001] The present invention relates to 3D interactive computer graphics, and more specifically, to arrangements and techniques for efficiently representing and storing vertex information for animation and display processing. Still more particularly, the invention relates to a 3D graphics integrated circuit including a vertex cache for more efficient imaging of 3D polygon data.

BACKGROUND AND SUMMARY OF THE INVENTION

[0002] Modern 3D computer graphics systems construct animated displays from display primitives, i.e., polygons. Each display object (e.g., a tree, a car, or a person or other character) is typically constructed from a number of individual polygons. Each polygon is represented by its vertices -- which together specify the location, orientation and size of the polygon in three-dimensional space --along with other characteristics (e.g., color, surface normals for shading, textures, etc.). Computer techniques can efficiently construct rich animated 3D graphical scenes using these techniques.

[0003] Low cost, high speed interactive 3D graphics systems such as video game systems are constrained in terms of memory and processing resources. Therefore, in such systems it is important to be able to efficiently represent and process the various polygons representing a display object. For example, it is desirable to make the data representing the display object compact, and to present the data to the 3D graphics system in a way so that all of the data needed for a particular task is conveniently available.

[0004] One can characterize data in terms of temporal locality and spatial locality. Temporal locality means the same data is being referenced frequently in a small amount of time. In general, the polygon-representing data for typical 3D interactive graphics applications has a large degree of temporal locality. Spatial locality means that the next data item referenced is stored close in memory to the last one referenced. Efficiency improvements can be realized by increasing the data's spatial locality. In a practical memory system that does not allow unlimited low-latency random access to an unlimited amount of data, performance is increased if all data needed to perform a given task is stored close together in low-latency memory.

[0005] To increase the spatial locality of the data, one can sort the polygon data based on the order of processing -- assuring that all of the data needed to perform a particular task will be presented at close to the same time so it can be stored together. For example, polygon data making up animations can be sorted in a way that is preferential to the type of animation being performed. As one example, typical complex interactive real-time animation such as surface deformation requires manipulation of all the vertices at the surfaces. To perform such animation efficiently, it is desirable to sort the vertex data in a certain way.

[0006] Typical 3D graphical systems perform animation processing and display processing separately, and these separate steps process the data differently. Unfortunately, the optimal order to sort the vertex data for animation processing is generally different from the optimal sort order for display processing. Sorting for animation may tend to add randomness to display ordering. By sorting a data stream to simplify animation processing, we make it harder to efficiently display the data.

[0007] Thus, for various reasons, it may not be possible to assume that spatial locality exists when accessing data for display. Difficulty arises from the need to efficiently access an arbitrarily large display object. In addition, for the reasons explained above, there will typically be some amount of randomness -- at least for display purposes -- in the order the vertex data as presented to the display engine. Furthermore, there may be other data locality above the vertex level that would be useful to implement (e.g., grouping together all polygons that share a certain texture).

[0008] One approach to achieving higher efficiency is to provide additional low-latency memory (e.g., the lowest latency memory system affordable). It might also be possible to fit a display object in fast local memory to achieve random access. However, objects can be quite large, and may need to be double-buffered. Therefore, the buffers required for such an approach could be very large. It might also be possible to use a main CPU's data cache to assemble and sort the polygon data in an optimal order for the display engine. However, to do this effectively, there would have to be some way to prevent the polygon data from thrashing the rest of the data cache. In addition, there would be a need to prefetch the data to hide memory latency -- since there will probably be some randomness in the way even data sorted for display order is accessed. Additionally, this approach would place additional loading on the CPU -- especially since there might be a need in certain implementations to assemble the data in a binary format the display engine can interpret. Using this approach, the main CPU and the display engine would become serial, with the CPU feeding the data directly to the graphics engine. Parallelizing the processing (e.g., to feed the display engine through a DRAM FIFO buffer) would require substantial additional memory access bandwidth as compared to immediate-mode feeding.

[0009] Thus, there exists a need for more efficient techniques that can be used to represent, store and deliver polygon data for a 3D graphics display process.

[0010] The present invention solves this problem by providing a vertex cache to organize indexed primitive vertex data streams.

[0011]   In accordance with one aspect provided by the present invention, polygon vertex data is fed to the display engine via a vertex cache. The vertex cache may be a small, low-latency memory that is local to (e.g., part of) display engine hardware. Flexibility and efficiency results from the cache providing a virtual memory view much larger than the actual cache contents.

[0012]   The vertex cache may be used to build up the vertex data needed for display processing on the fly on an as-needed basis. Thus, rather than pre-sorting the vertex data for display purposes, the vertex cache can simply fetch the relevant blocks of data on an as-needed basis to make it available to the display processor. We have found that based on the high degree of temporal locality exhibited by the vertex data for interactive video game display and the use of particularly optimal indexed-array data structures (see below), most of the vertex data needed at any given time will be available in even a small set-associative vertex cache having a number of cache lines proportional to the number of vertex data streams. One example optimum arrangement provides a 512 x 128-bit dual ported RAM to form an 8 set-associative vertex cache.

[0013]   Efficiency can be increased by customizing and optimizing the vertex cache and associated tags for the purpose of delivering vertices to the graphics engine --allowing more efficient prefetching and assembling of vertices than might be possible using a general-purpose cache and tag structure. Because the vertex cache allows data to be fed directly to the display engine, the cost of additional memory access bandwidth is avoided. Direct memory access may be used to efficiently transfer vertex data into the vertex cache.

[0014]   To further increase the efficiencies afforded by the vertex cache, it is desirable to reduce the need to completely re-specify a particular polygon or set of polygons each time it is (they are) used. In accordance with a further aspect provided by the present invention, polygons can be represented as arrays, e.g., linear lists of data components representing some feature of a vertex (for example, positions, colors, surface normals, or texture coordinates). Each display object may be represented as a collection of such arrays along with various sets of indices. The indices reference the arrays for a particular animation or display purpose. By representing polygon data as indexed component lists, discontinuities are allowed between mappings. Further, separating out individual components allows data to be stored more compactly (e.g., in a fully compressed format). The vertex cache provided by the present invention can accommodate streams of such indexed data up to the index size.

[0015]   Through use of an indexed vertex representation in conjunction with the vertex cache, there is no need to provide any resorting for display purposes. For example, the vertex data may be presented to the display engine in a order presorted for animation as opposed to display -- making animation a more efficient process. The vertex cache uses the indexed vertex data structure representation to efficiently make the vertex data available to the display engine without any need for explicit resorting.

[0016]   Any vertex component can be index-referenced or directly inlined in the command stream. This enables efficient data processing by the main processor without requiring the main processor's output to conform to the graphics display data structure. For example, lighting operations performed by the main processor may generate only a color array from a list of normals and positions by loop-processing a list of lighting parameters to generate the color array. There is no need for the animation process to follow a triangle list display data structure, nor does the animation process need to reformat the data for display. The display process can naturally consume the data provided by the animation process without adding substantial data reformatting overhead to the animation process.

[0017]   On the other hand, there is no penalty for sorting the vertex data in display order; the vertex data is efficiently presented to the display engine in either case, without the vertex cache significantly degrading performance vis-a-vis a vertex presentation structure optimized for presenting data presorted for display.

[0018]   In accordance with a further aspect provided by this invention, the vertex data includes quantized, compressed data streams in any of several different formats (e.g., 8-bit fixed point, 16-bit fixed point, or floating point). This data can be indexed (i.e., referenced by the vertex data stream) or direct (i.e., contained within the stream itself). These various data formats can all be stored in the common vertex cache, and subsequently decompressed and converted into a common format for the graphics display pipeline. Such hardware support of flexible types, formats and numbers of attributes as either immediate or indexed input data avoids complex and time-consuming software data conversion.

BRIEF DESCRIPTION OF THE DRAWINGS

[0019]   These and other features and advantages provided by the present invention will be better and more completely understood by referring to the following detailed description of preferred embodiments in conjunction with the drawings of which:

Figure 1 is a block diagram of an example interactive 3D graphics system;
Figure 1A is a block diagram of the example graphics and audio coprocessor shown in Figure 1;
Figure 1B is a more detailed schematic diagram of portions of the Figure 1A graphics and audio coprocessor showing an example 3D pipeline graphics processing arrangement;
Figure 2 shows an example command processor including a vertex cache provided with vertex index

array data;

Figure 2A shows an example display list processor including a vertex cache provided in accordance with the present invention;

Figure 2B shows an example dual FIFO arrangement;

Figure 3 is a schematic diagram of an example indexed vertex data structure;

Figure 3A shows an example vertex descriptor block;

Figure 4 is a block diagram of an example vertex cache implementation;

Figure 5 shows an example vertex cache memory address format; and

Figure 6 shows an example vertex cache tag status register format.

DETAILED DESCRIPTION OF PRESENTLY PRE-FERRED EXAMPLE EMBODIMENTS

[0020] Figure 1 is a schematic diagram of an overall example interactive 3D computer graphics system 100 in which the present invention may be practiced. System 100 can be used to play interactive 3D video games accompanied by interesting stereo sound. Different games can be played by inserting appropriate storage media such as optical disks into an optical disk player 134. A game player can interact with system 100 in real time by manipulating input devices such as handheld controllers 132, which may include a variety of controls such as joysticks, buttons, switches, keyboards or keypads, etc.

[0021] System 100 includes a main processor (CPU) 102, a main memory 104, and a graphics and audio coprocessor 106. In this example, main processor 102 receives inputs from handheld controllers 132 (and/or other input devices) via coprocessor 100. Main processor 102 interactively responds to such user inputs, and executes a video game or other graphics program supplied, for example, by external storage 134. For example, main processor 102 can perform collision detection and animation processing in addition to a variety of real time interactive control functions.

[0022] Main processor 102 generates 3D graphics and audio commands and sends them to graphics and audio coprocessor 106. The graphics and audio coprocessor 106 processes these commands to generate interesting visual images on a display 136 and stereo sounds on stereo loudspeakers 137R, 137L or other suitable sound-generating devices.

[0023] System 100 includes a TV encoder 140 that receives image signals from coprocessor 100 and converts the image signals into composite video signals suitable for display on a standard display device 136 (e.g., a computer monitor or home color television set). System 100 also includes an audio codec (compressor/decompressor) 138 that compresses and decompresses digitized audio signals (and may also convert between digital and analog audio signalling formats). Audio codec 138 can receive audio inputs via a buffer 140 and provide them to coprocessor 106 for processing (e.g., mixing with other audio signals the coprocessor generates and/or receives via a streaming audio output of optical disk device 134). Coprocessor 106 stores audio related information in a memory 144 that is dedicated to audio tasks. Coprocessor 106 provides the resulting audio output signals to audio codec 138 for decompression and conversion to analog signals (e.g., via buffer amplifiers 142L, 142R) so they can be played by speakers 137L, 137R.

[0024] Coprocessor 106 has the ability to communicate with various peripherals that may be present within system 100. For example, a parallel digital bus 146 may be used to communicate with optical disk device 134. A serial peripheral bus 148 may communicate with a variety of peripherals including, for example, a ROM and/or real time clock 150, a modem 152, and flash memory 154. A further external serial bus 156 may be used to communicate with additional expansion memory 158 (e.g., a memory card).

Graphics And Audio Coprocessor

[0025] Figure 1A is a block diagram of components within coprocessor 106. Coprocessor 106 may be a single integrated circuit. In this example, coprocessor 106 includes a 3D graphics processor 107, a processor interface 108, a memory interface 110, an audio digital signal processor (DSP) 162, an audio memory interface (I/F) 164, an audio interface and mixer 166, a peripheral controller 168, and a display controller 128.

[0026] 3D graphics processor 107 performs graphics processing tasks, and audio digital signal processor 162 performs audio processing tasks. Display controller 128 accesses image information from memory 104 and provides it to TV encoder 140 for display on display device 136. Audio interface and mixer 166 interfaces with audio codec 138, and can also mix audio from different sources (e.g., a streaming audio input from disk 134, the output of audio DSP 162, and external audio input received via audio codec 138). Processor interface 108 provides a data and control interface between main processor 102 and coprocessor 106. Memory interface 110 provides a data and control interface between coprocessor 106 and memory 104. In this example, main processor 102 accesses main memory 104 via processor interface 108 and memory controller 110 that are part of coprocessor 106. Peripheral controller 168 provides a data and control interface between coprocessor 106 and the various peripherals mentioned above (e.g., optical disk device 134, controllers 132, ROM and/or real time clock 150, modem 152, flash memory 154, and memory card 158). Audio memory interface 164 provides an interface with audio memory 144.

[0027] Figure 1B shows a more detailed view of 3D

graphics processor 107 and associated components within coprocessor 106. 3D graphics processor 107 includes a command processor 114 and a 3D graphics pipeline 116. Main processor 102 communicates streams of graphics data (i.e., display lists) to command processor 114. Command processor 114 receives these display commands and parses them (obtaining any additional data necessary to process them from memory 104), and provides a stream of vertex commands to graphics pipeline 116 for 3D processing and rendering. Graphics pipeline 116 generates a 3D image based on these commands. The resulting image information may be transferred to main memory 104 for access by display controller 128 -- which displays the frame buffer output of pipeline 116 on display 136.

[0028]   In more detail, main processor 102 may store display lists in main memory 104, and pass pointers to command processor 114 via bus interface 108. The command processor 114 (which includes a vertex cache 212 discussed in detail below) fetches the command stream from CPU 102, fetches vertex attributes from the command stream and/or from vertex arrays in memory, converts attribute types to floating point format, and passes the resulting complete vertex polygon data to the graphics pipeline 116 for rendering/rasterization. As explained in more detail below, vertex data can come directly from the command stream, and/or from a vertex array in memory where each attribute is stored in its own linear array. A memory arbitration circuitry 130 arbitrates memory access between graphics pipeline 116, command processor 114 and display unit 128. As explained below, an on-chip 8-way set-associative vertex cache 212 is used to reduce vertex attribute access latency.

[0029]   As shown in Figure 1B, graphics pipeline 116 may include transform unit 118, a setup/rasterizer 120, a texture unit 122, a texture environment unit 124 and a pixel engine 126. In graphics pipeline 116, transform unit 118 performs a variety of 3D transform operations, and may also perform lighting and texture effects. For example, transform unit 118 transforms incoming geometry per vertex from object space to screen space; transforms incoming texture coordinates and computes projective texture coordinates; performs polygon clipping; performs per vertex lighting computations; and performs bump mapping texture coordinate generation. Set up/rasterizer 120 includes a set up unit which receives vertex data from the transform unit 118 and sends triangle set up information to rasterizers performing edge rasterization, texture coordinate rasterization and color rasterization. Texture unit 122 performs various tasks related to texturing, including multi-texture handling, post-cache texture decompression, texture filtering, embossed bump mapping, shadows and lighting through the use of projective textures, and BLIT with alpha transparency and depth. Texture unit 122 outputs filtered texture values to the texture environment unit 124. Texture environment unit 124 blends the polygon color and texture color together, performing texture fog and other environment-related functions. Pixel engine 126 performs z buffering and blending, and stores data into an on-chip frame buffer memory.

[0030]   Thus, graphics pipeline 116 may include one or more embedded DRAM memories (not shown) to store frame buffer and/or texture information locally. The on-chip frame buffer is periodically written to main memory 104 for access by display unit 128. The frame buffer output of graphics pipeline 116 (which is ultimately stored in main memory 104) is read each frame by display unit 128. Display unit 128 provides digital RGB pixel values for display on display 136.

Vertex Cache And Vertex Index Array

[0031]   Figure 2 is a schematic illustration of command processor 114 including a vertex cache 212 and a display list processor 213. Command processor 114 handles a wide range of vertex and primitive data structures, from a single stream of vertex data containing position, normal, texture coordinates and colors to fully indexed arrays. Any vertex component can be index-referenced or directly inlined in the command stream. Command processor 114 thus supports flexible types, formats and numbers of attributes as either immediate or indexed data.

[0032]   Display list processor 213 within command processor 114 processes display list commands provided by CPU 102 -- typically via a buffer allocated within main memory 104. Vertex cache 212 caches indexed polygon vertex data structures such as the example data structure 300 shown in Figure 2. Example indexed polygon vertex data structure 300 may include a vertex index array 304 which references a number of vertex component data arrays (e.g., a color data array 306a, a texture vertex data array 306b, a surface normal data array 306c, a position vertex data array 306d, and so on). Vertex cache 212 accesses the vertex data from these arrays 306 in main memory 104, and caches them for fast access and use by display list processor 213.

Display List Processor

[0033]   Figure 2A shows example display list processor 213 performed by command processor 114. In this Figure 2A example, display list processor 213 provides several stages of parsing. Display list commands received from main processor 102 are interpreted by a display list stream parser 200. Display list stream parser 200 may use an address stack 202 to provide nesting of instructions -- or dual FIFOS may be used to store a stream of vertex commands from a FIFO in main memory 106 to allow subroutine branching in instancing (see Figure 2B) without need for reloading prefetched vertex command data. Using the Figure 2B approach, the display list commands may thus provide for a one-level-

deep display list -- where the top level command stream can call the display list one level deep. This "call" capability is useful for pre-computed commands and instancing in geometry.

[0034]   Display list stream parser 200 routes commands that affect the state of graphics pipeline 116 to the graphics pipeline. The remaining primitive command stream is parsed by a primitive stream parser 204 based on a primitive descriptor obtained from memory 104 (see below).

[0035]   The indices to vertices are de-referenced and parsed by a vertex stream parser 208 based on a vertex descriptor 306 which may be provided in a table in hardware. The vertex stream provided to vertex stream parser 208 may include such indices to vertex data stored within main memory 104. Vertex stream parser 208 can access this vertex data from main memory 104 via vertex cache 212 --thus separately providing the vertex commands and associated referenced vertex attributes via different paths in the case of indexed as opposed to direct data. In one example, vertex stream parser 208 addresses vertex cache 212 as if it were the entirety of main memory 104. Vertex cache 212, in turn, retrieves (and often times, may prefetch) vertex data from main memory 104, and caches it temporarily for use by vertex stream parser 208. Caching the vertex data in vertex cache 212 reduces the number of accesses to main memory 104 -- and thus the main memory bandwidth required by command processor 114.

[0036]   Vertex stream parser 208 provides data for each vertex to be rendered within each triangle (polygon). This per-vertex data is provided, along with the per-primitive data outputted by primitive stream parser 204, to a decompression/inverse quantizer block 214. Inverse quantizer 214 converts different vertex representations (e.g., 8-bit and 16-bit fixed point format data) to a uniform floating-point representation used by graphics pipeline 116. Inverse quantizer 214 provides hardware support for a flexible variety of different types, formats and numbers of attributes, and such data can be presented to display list processor 213 as either immediate or indexed input data. The uniform floating-point representation output of inverse quantizer 214 is provided to graphics pipeline 116 for rasterization and further processing. If desired as an optimization, a further small cache or buffer may be provided at the output of inverse quantizer 214 to avoid the need to re-transform vertex strip data.

Vertex Index Array

[0037]   Figure 3 shows a more detailed example of an indexed vertex list 300 of the preferred embodiment used to provide indirect (i.e., indexed) vertex attribute data via vertex cache 212. This generalization indexed vertex list 300 may be used to define primitives in the system shown in Figure 1. Each primitive is described by a list of indices, each of which indexes into an array of vertices. Vertices and primitives each use format descriptors to define the types of their items. These descriptors associate an attribute with a type. An attribute is a data item that has a specific meaning to the rendering hardware. This affords the possibility of programming the hardware with descriptors so it can parse and convert the vertex/primitive stream as it is loaded. Using the minimum size type and the minimum number of attributes per vertex leads to geometry compression. The Figure 3 arrangement also allows attributes to be associated with the vertices, the indices, or the primitive, as desired.

[0038]   Thus, in the Figure 3 example indexed vertex array 300, a primitive list 302 defines each of the various primitives (e.g., triangles) in the data stream (e.g., prim0, prim1, prim2, prim3, ...). A primitive descriptor block 308 may provide attributes common to a primitive (e.g., texture and connectivity data which may be direct or indexed). Each primitive within primitive list 302 indexes corresponding vertices within a vertex list 304. A single vertex within vertex list 304 may be used by multiple primitives within primitive list 302. If desired, primitive list 302 may be implied rather than explicit -- i.e., vertex list 304 can be ordered in such a way as to define corresponding primitives by implication (e.g., using triangle strips).

[0039]   A vertex descriptor block 306 may be provided for each vertex within vertex list 304. Vertex descriptor block 306 includes attribute data corresponding to a particular vertex (e.g., rgb or other color data, alpha data, xyz surface normal data). As shown in Figure 2, vertex descriptor block 306 may comprise a number of different indexed component blocks. The vertex attribute descriptor block 306 defines which vertex attributes are present, the number and size of the components, and how the components are referenced (e.g., either direct -- that is, included within the quantized vertex data stream -- or indexed). In one example, the vertices in a DRAW command for a particular primitive all have the same vertex attribute data structure format.

[0040]   Figure 3A shows an example list of attributes provided by vertex attribute block 306. The following attributes may be provided:

Attribute
Position
Normal
Diffused Color
Specular Color
(Skinning)
Texture 0 Coordinate

(continued)

Attribute
Texture 1 Coordinate
Texture 2 Coordinate
Texture 3 Coordinate
Texture 4 Coordinate
Texture 5 Coordinate
Texture 6 Coordinate
Texture 7 Coordinate

[0041] In this example vertex attribute descriptor block 306, the position attribute is always present, may be either indexed or direct, and can take a number of different quantized, compressed formats (e.g., floating point, 8-bit, Integer, or 16-bit). All remaining attributes may or may not be present for any given vertex, and may be either indexed or direct as desired. The texture coordinate values may, like the position values, be represented in a variety of different formats (e.g., 8-bit integer, 16-bit integer or floating point), as can the surface normal attribute. The diffused and specular color attributes may provide 3 (rgb) or 4 (rgba) values in a variety of formats including 16-bit threes-complement, 24-bit threes-complement, 32-bit threes-complement, or 16-, 24- or 32-bit fours-complement representations). All vertices for a given primitive preferably have the same format.

[0042] In this example, vertex descriptor 306 references indexed data using a 16-bit pointer into an array of attributes. A particular offset used to access a particular attribute within the array depends upon a number of factors including, e.g., the number of components in the attribute; the size of the components, padding between attributes for alignment purposes; and whether multiple attributes are interleaved in the same array. A vertex can have direct and indirect attributes intermixed, and some attributes can be generated by the hardware (e.g., generating a texture coordinate from a position). Any attribute can be sent either directly or as an index into an array. Vertex cache 212 includes sufficient cache lines to handle the typical number of respective data component streams (e.g., position, normal, color and texture) without too many cache misses.

Vertex Cache Implementation

[0043] Figure 4 shows an example schematic diagram of vertex cache 212 and associated logic. Vertex cache 212 in this example includes an 8-Kilobyte cache memory 400 organized as a 512 x 112-bit dual ported RAM. Since there are multiple attribute streams being looked up in the cache 212, an eight set-associative cache including eight tag lines 402 is used to reduce thrashing. Each tag line includes a 32 x 16 bit dual

ported tag RAM 404 and associated tag status register 406. Tag RAMS 404 store the main memory address of the corresponding data block stored within vertex RAM 400. Address calculation block 408 determines whether necessary vertex attribute data is already present within vertex RAM 400 -- or whether an additional fetch to main memory is required. Cache lines are prefetched from main memory 104 to hide memory latency. Data required to process a particular component is stored within a queue 410 having a depth that is proportional to memory latency.

[0044] Figure 5 shows an example memory address format provided by vertex stream parser 208 to vertex cache 212. This memory address 450 includes a field 452 providing a byte offset into a cache line; a tag RAM address 454; and a main memory address for comparison with the contents of tag RAMs 404. Address calculation block 408 compares the main memory address 456 with the tag RAM 404 contents to determine whether the required data is already cached within vertex RAM 400, or whether it needs to be fetched from main memory 104.

[0045] The tag status registers 406 store data in the format shown in Figure 6. A "data valid" field 462 indicates whether the data in that particular cache line is valid. A counter field 464 keeps track of the number of entries in queue 410 that depend on the cache line. Counter field 464 is used in the case that all tag status registers 406 show "data valid" if a miss occurs. Address calculation block 408 then needs to throw one of the cache lines out to make room for the new entry. If counter field 464 is not zero, the cache line is still in use and cannot be thrown away. Based on a modified partial LRU algorithm, address calculation block 408 selects one of the cache lines for replacement. The "data valid" field 462 is set to "invalid", and the cache line is replaced with a new contents from main memory 104. If another attribute index maps to the same cache line, the counter field 464 is incremented. Once the data arrives from main memory 104, the "data valid" bit is set and an entry can be processes from queue 410. Otherwise, the processing of queue 410 will be stalled until the data gets into the cache RAM 400. Once the cache RAM 400 is accessed for the queue entry, counter 464 decrements.

[0046] While the invention has been described in connection with what is presently considered to be the most practical and preferred embodiment, it is to be understood that the invention is not to be limited to the disclosed embodiment, but on the contrary, is intended to cover various modifications and equivalent arrangements included within the scope of the appended claims.

Claims

1. In a 3D videographics system including a memory storing polygon vertex data, and a 3D graphics

engine that generates and displays images at least in part in response to said polygon vertex data, an improvement comprising a vertex cache arrangement operatively coupled to said memory and to said 3D graphics engine, said vertex cache arrangement caching said vertex data from said memory for use by said 3D graphics engine, wherein said polygon vertex data includes an indexed vertex data representation, and said vertex cache arrangement operates in response to said indexed vertex data representation to cache indexed vertex data and make it available for efficient access by the 3D graphics engine without requiring explicit resorting of said polygon vertex data for image display.

2. A vertex cache arrangement as in claim 1 wherein 3D graphics engine is disposed on an integrated circuit, and said vertex cache arrangement comprises a memory device disposed on said integrated circuit and operatively coupled to said 3D graphics engine.

3. A vertex cache arrangement as in claim 2 wherein said memory device comprises a set-associative cache memory for caching said vertex data.

4. A vertex cache arrangement as in claim 3 wherein said set-associative cache memory provides eight cache lines.

5. A vertex cache arrangement as in claim 2 wherein said memory device comprises an 8 kilobyte low-latency random access memory.

4. A vertex cache arrangement as in claim 1 including plural cache tag lines.

5. A vertex cache arrangement as in claim 1 including plural cache tag status registers.

6. A vertex cache arrangement as in claim 1 including a queue and a miss queue.

7. A vertex cache arrangement as in claim 1 wherein said vertex cache arrangement is coupled to said memory and, in use, fetches vertex data from said memory as said vertex data is needed by said 3D graphics engine.

8. A vertex cache arrangement as in claim 1 further including a hardware-based inverse quantizer operatively coupled between said vertex cache and said 3D graphics engine, said inverse quantizer converting plural vertex data formats into a uniform floating-point format for consumption by said 3D graphics engine.

9. A vertex cache arrangement as in claim 1 wherein said indexed vertex data representation comprises an indexed array referencing attribute data.

10. A vertex cache arrangement as in claim 9 wherein said indexed array directly references said attribute data.

11. A vertex cache arrangement as in claim 9 wherein said indexed array indirectly references said attribute data.

12. In a 3D videographics system including a memory storing polygon vertex data, and a 3D graphics engine that generates and displays images at least in part in response to said polygon vertex data, an improvement comprising a method for accessing said vertex data from said memory for use by said 3D graphics engine, said method including the steps of representing said polygon vertex data in an indexed vertex data representation, and caching said indexed vertex data in a low-latency cache memory device local to said 3D graphics engine to make said indexed vertex data available for efficient access by the 3D graphics engine.

13. A method as in claim 12 further including the step of fetching vertex data from said memory to said cache memory device as said vertex data is needed by said 3D graphics engine.

14. A method as in claim 12 further including the step of converting plural vertex data formats stored in said cache memory device into a uniform floating-point format for consumption by said 3D graphics engine.
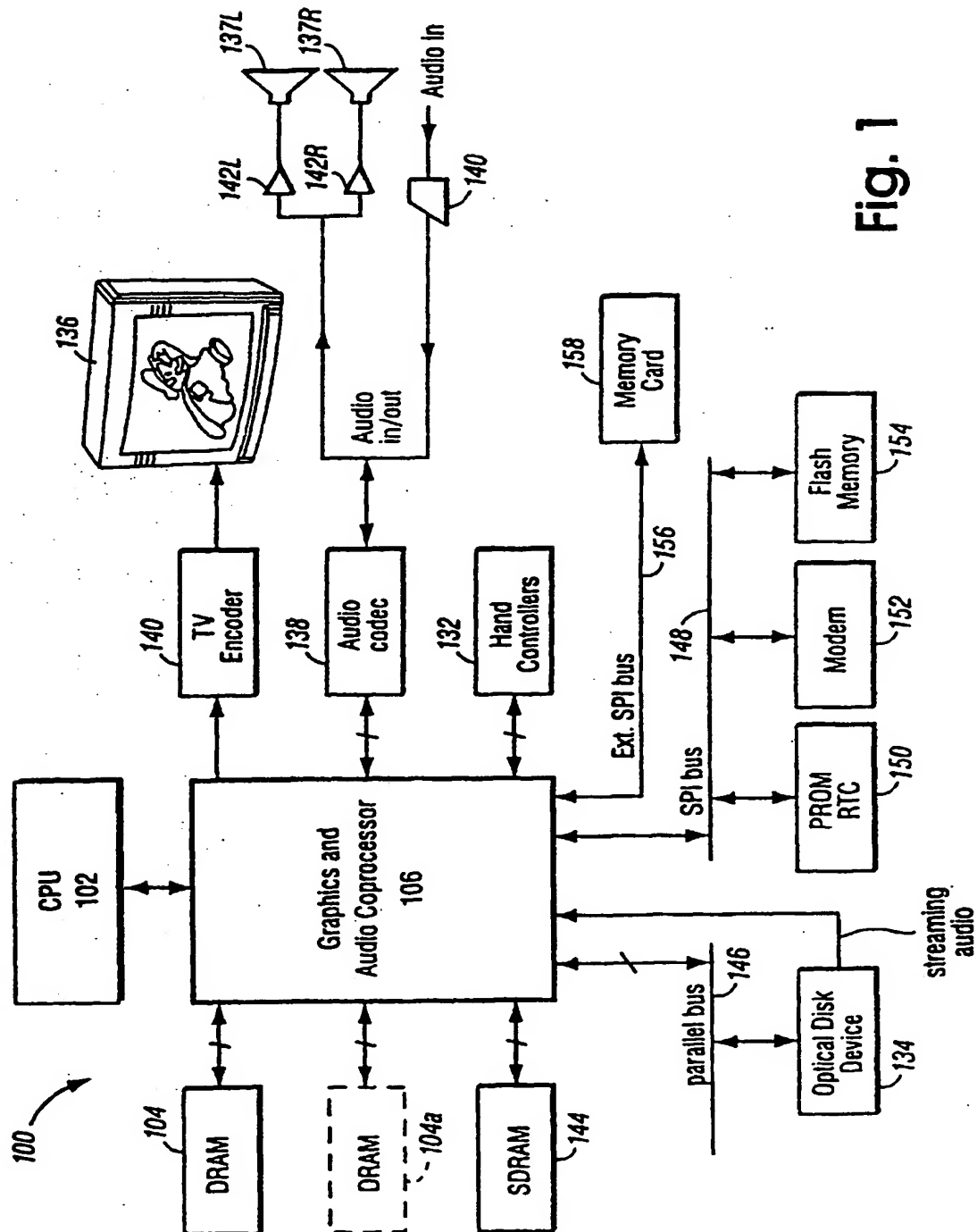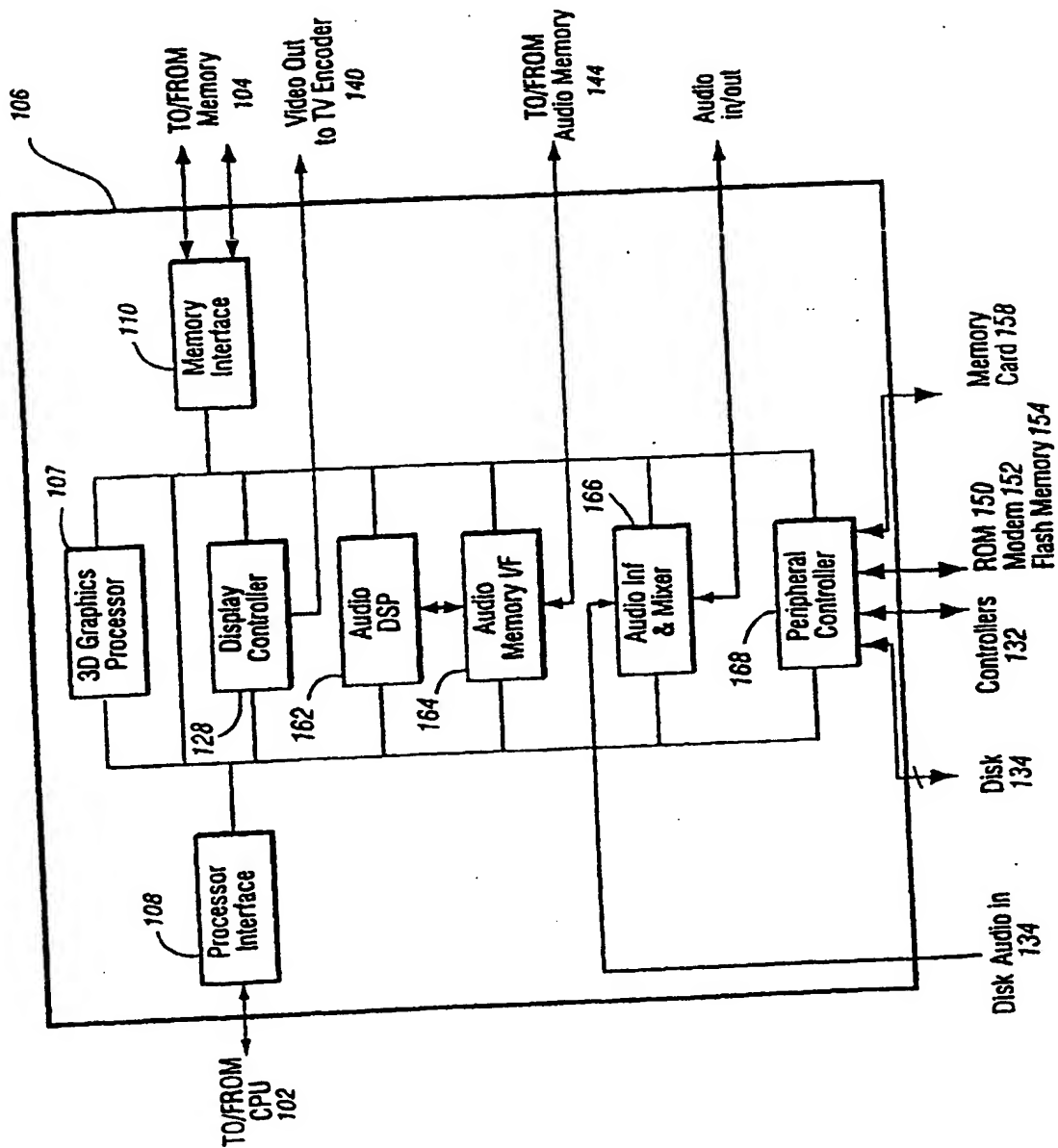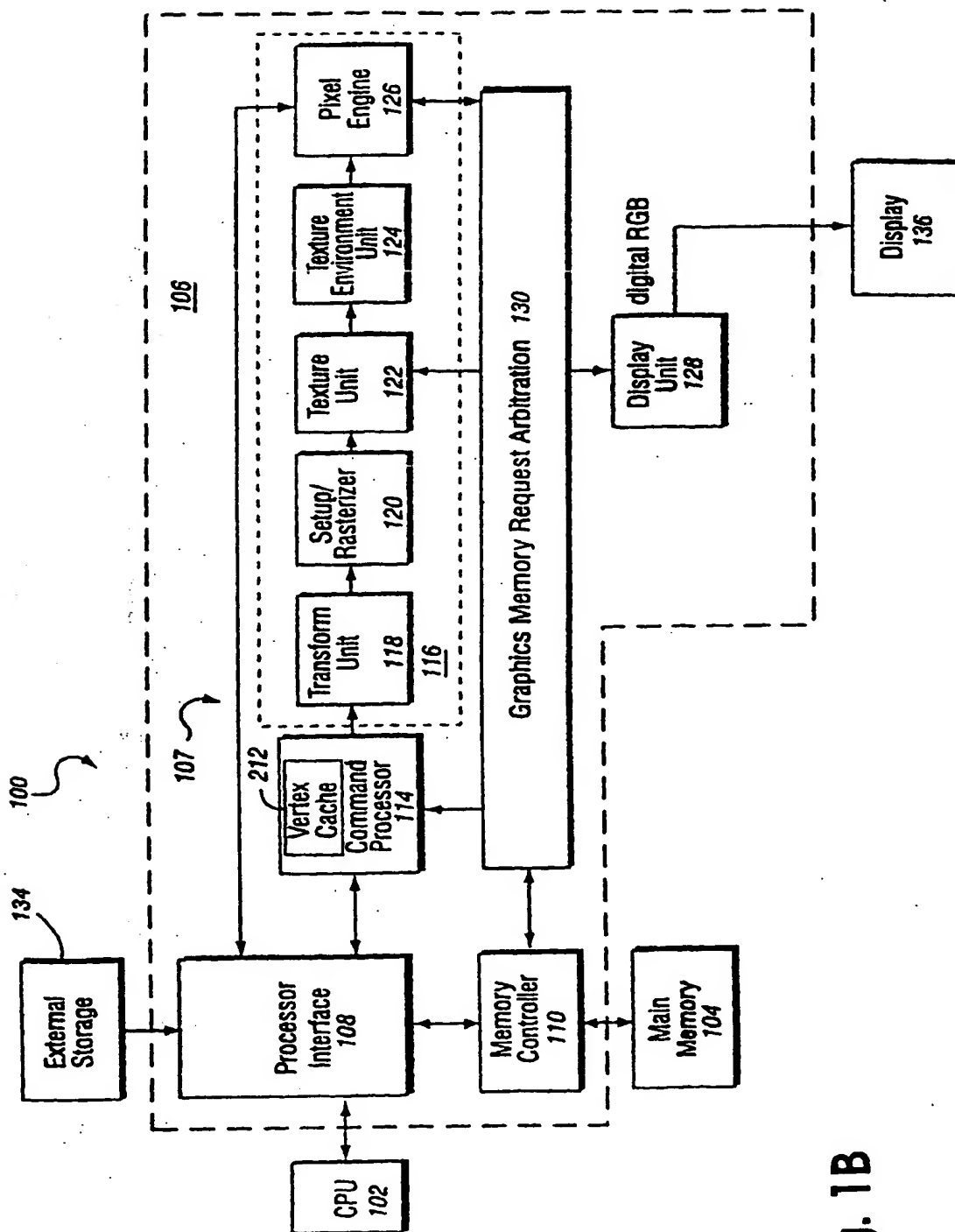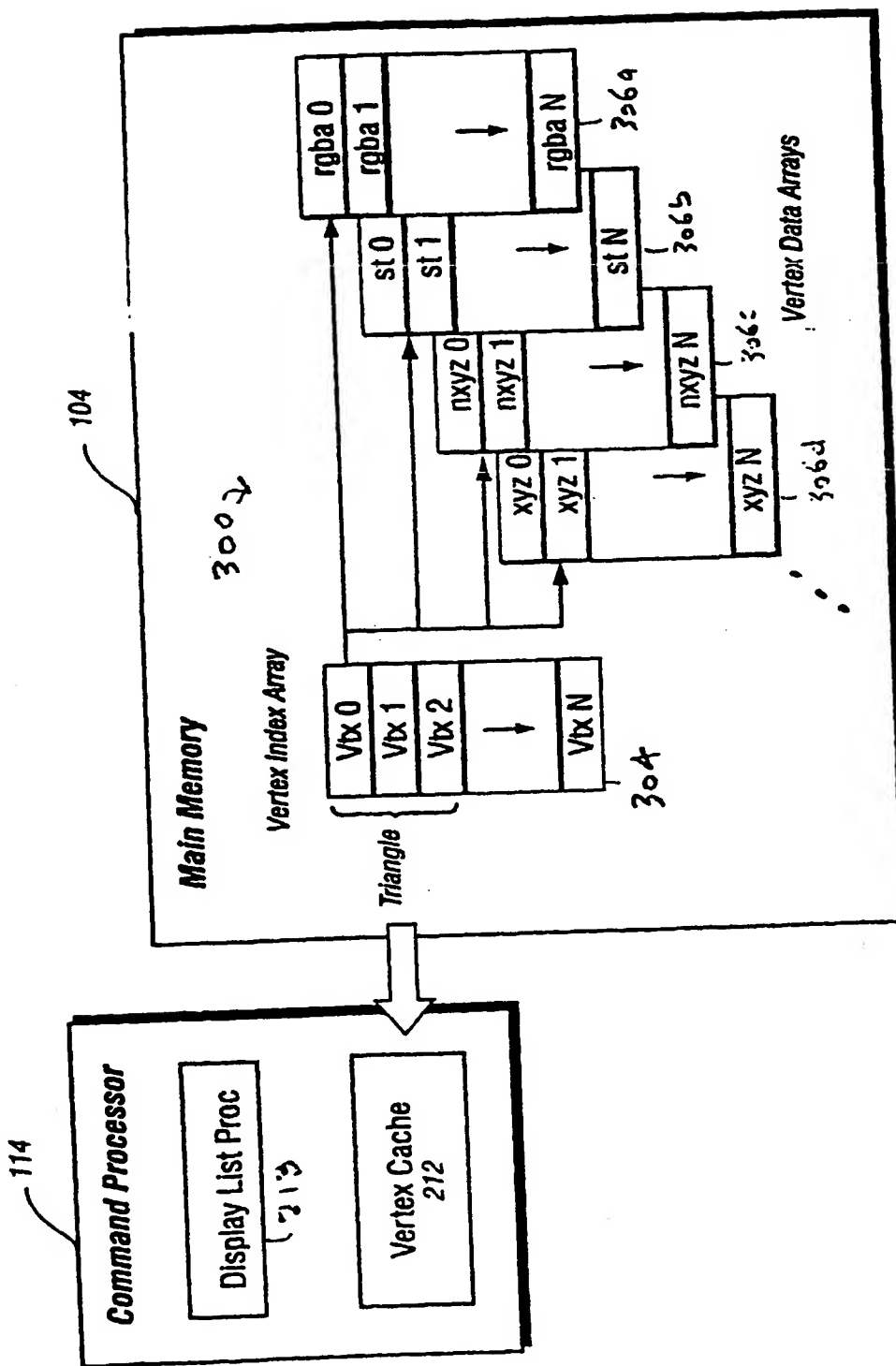
Fig. 1

Fig. 1A

Fig. 1B

Fig. 2

Command Stream

213

addr   data

114

202

Address Stack — push / pop → Display List Stream Parser   200

Draw command

→ pipeline state

Primitive Descriptor 308 → Primitive Stream Parser   204

addr

primitive stream

→ per-primitive/per-index data

vertex stream

208

Vertex Descriptor 306 → Vertex Stream Parser

addr → Vertex Cache   addr → **Vertex Components from Main Memory** 104

data ← data ←

212

per vertex data

new triangle

primitive data

214

Inverse Q

To display pipeline 116

Fig. 2A

Fig. 2B

Fig. 3

Vertex Descriptor

| r u8 | g u8 | b u8 | a u8 | x s16 | y s16 | z s16 | pad s16 |

306

Vertex List

| vert 0 | vert 1 | vert 2 | vert 3 | vert 4 | vert 5 | vert 6 | vert 7 | vert 8 | vert 9 | vert 10 | vert 11 |

each vert has this structure

304

Primitive List

| prim 0 | prim 1 | prim 2 | prim 3 |

302

primitive list indexes into vertex list

Primitive Descriptor

| s u16 | t u16 | indx u8 | s u16 | t u16 | indx u8 | s u16 | t u16 | indx u8 | pad u8 |

308

each primitive has this format

300

306

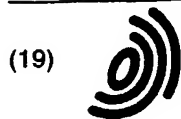| Position |
| Normal |
| Color-Diffused |
| Color-Specular |
| Texture 0 Coordinates |
| Texture 1 Coordinates |
| Texture 2 Coordinates |

Fig. 3A

Vertex Attribute Table

Fig. 4

To data conversion unit 214

450

Compared with Tag RAM Contents	Tag RAM Address	Byte Offset Into Cache Line

456

454

452

Fig. 5

406

Data Valid	Counter

462

464

Fig. 6

• Van Hook, Timothy J.
Atherton, CA 94027 (US)
• Cheng, Howard H.
Redmond, Washington 98053 (US)
• Moore, Robert
Redmond, Washington 98053 (US)

(74) Representative:
Horton, Andrew Robert Grant et al
BOWLES HORTON
Felden House
Dower Mews
High Street
Berkhamsted Hertfordshire HP4 2BL (GB)

(54) Vertex cache for 3D computer graphics

(57) In a 3D interactive computer graphics system such as a video game display system, polygon vertex data is fed to the display engine via a vertex cache used to cache and organize indexed primitive vertex data streams. The vertex cache may be a small, low-latency cache memory local to the display engine hardware. Polygons can be represented as indexed arrays, e.g., indexed linear lists of data components representing some feature of a vertex (for example, positions, colors, surface normals, or texture coordinates). The vertex cache can fetch the relevant blocks of indexed vertex attribute data on an as-needed basis to make it available to the display processor -- providing spatial locality for display processing without requiring the vertex data to be prestored in display order. Efficiency can be increased by customizing and optimizing the vertex cache and associated tags for the purpose of delivering vertices to the graphics engine -- allowing more efficient prefetching and assembling of vertices than might be possible using a general-purpose cache and tag structure.
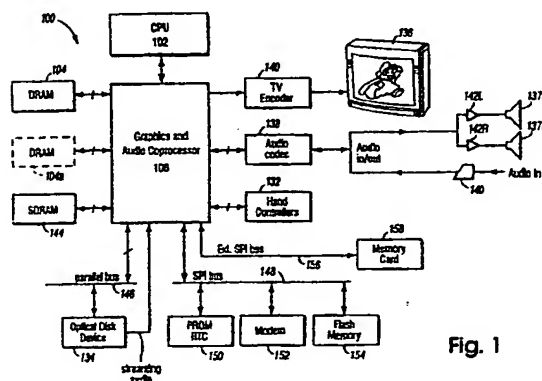
Fig. 1

EP 1 096 427 A3

European Patent Office

EUROPEAN SEARCH REPORT

Application Number

EP 00 30 9315

DOCUMENTS CONSIDERED TO BE RELEVANT

Category	Citation of document with indication, where appropriate, of relevant passages	Relevant to claim	CLASSIFICATION OF THE APPLICATION (Int.Cl.7)
X	HOPPE: "Optimization of mesh locality for transparent vertex caching" PROCEEDINGS OF SIGGRAPH , 8 - 13 August 1999, pages 269-276, XP000981469 * page 269, left-hand column, line 1 - page 270, left-hand column, line 9 *	1,2,6-14	G06T15/00
A	WO 93 04429 A (UNIV WASHINGTON) 4 March 1993 (1993-03-04) * page 4, line 17 - page 5, line 14 * * page 12, line 3 - page 15, line 36 *	3-6	

TECHNICAL FIELDS SEARCHED (Int.Cl.7)

G06T

The present search report has been drawn up for all claims

Place of search	Date of completion of the search	Examiner
BERLIN	20 February 2003	Burgaud, C

CATEGORY OF CITED DOCUMENTS

X : particularly relevant if taken alone
Y : particularly relevant if combined with another document of the same category
A : technological background
O : non-written disclosure
P : intermediate document

T : theory or principle underlying the invention
E : earlier patent document, but published on, or after the filing date
D : document cited in the application
L : document cited for other reasons

& : member of the same patent family, corresponding document

EPO FORM 1503 03.82 (P04C01)

ANNEX TO THE EUROPEAN SEARCH REPORT
ON EUROPEAN PATENT APPLICATION NO.

EP 00 30 9315

This annex lists the patent family members relating to the patent documents cited in the above-mentioned European search report.
The members are as contained in the European Patent Office EDP file on
The European Patent Office is in no way liable for these particulars which are merely given for the purpose of information.

20-02-2003

Patent document cited in search report		Publication date	Patent family member(s)		Publication date
WO 9304429	A	04-03-1993	EP	0739513 A1	30-10-1996
			JP	6509893 T	02-11-1994
			WO	9304429 A2	04-03-1993
			US	5467459 A	14-11-1995